A Semantic-based Cache Replacement Algorithm for Mobile File Access

Sharun Santhosh and Weisong Shi

Department of Computer Science
Wayne State University
{*sharun, weisong*}*@wayne.edu*

Abstract

The ability to stay continually *connected*, due to the ubiquitous availability of a wide variety of networks, is a present day reality. Yet the convenience and transparency of distributed file systems across heterogeneous networks have a long way to go. In this paper, we examined the cache replacement algorithm on the client-side by taking the low-bandwidth connection into consideration. Motivated by the file access patterns in distributed file systems, we have proposed a novel caching replacement algorithm that takes both inter- and intra- file semantics into consideration when making caching replacement decisions. Not only does our approach perform better than conventional algorithms in terms of the hit rate, but also it reduces the number of cache replacements significantly, which results in a big saving on synchronization-related communication overhead, essential to the effectiveness of mobile file access.

1 Introduction

The dream of staying connected anywhere, anytime isn't any more a foreseeable eventuality but has become a present day reality. A well traveled laptop user may stay connected via half a dozen different networks throughout the course of a day, from a cable modem or DSL connection at home, to a high speed ethernet network at work or school, to a bluetooth network in the car, to a WiFi network at the airport or the neighborhood coffee shop.

Along with the freedom of moving, a user expects to be connected to his/her personal files and data whereever he/she may be. Each type of network has its own characteristic requirements, services and limitations. To be effective, the file system should be able to adapt to the variations of the underlying network available, as exploited in the Cegor file system [11].

Traditionally, the three basic steps involved in accessing data on the road are: (1) retrieve the files from the server, (2) work on them locally, and (3) write the changes back to the server. Almost all clients in distributed file systems have a client-side cache to *minimize communication with the server*, improving system performance on the whole. It can not be assumed that a fast network connection to a file server *always* exists, which is a fact that will not be changed in the foreseeable future. Thus policies must be designed to have a minimal reliance on the underlying communication overhead, resulting from cache misses (fetching) and cache replacement (update synchronization).

In the most popular distributed file system NFS [3], where delayed write back is implemented, writes are flushed back to the server from the cache after some preset time or when a file is closed. The Andrew file system AFS [5] uses a write back on close policy to reduce write traffic to the server. When the cache is full and programs request more data from AFS, the Cache Manager must flush out cache chunks to make room for the data based on a LRU type algorithm. So each time when a cache gets full, data has to be either *written back* to the server or dropped, which generates an *additional exchange* between the client and server. Therefore, communication overhead increases with replacements. Caches for distributed file systems, that operate across heterogeneous, especially low-bandwidth, networks, must not only provide a high hit rate, which is the same goal as of conventional caches that operate over homogeneous networks, but also perform as few replacements as possible.

With this in mind, in this paper we propose a semantic-based cache replacement algorithm, which takes both intra-file and inter-file relationships into consideration. File access patterns aren't random, and have been exploited by several algorithms that utilize the inherent relationship (which we will define shortly) between files to perform pre-fetching [1, 2] for disconnect operations. Our approach, from a different angle, does not use this relationship information to fetch files, rather it concentrates its efforts on keeping these relationships in the client cache. An *eviction index* is defined and calculated based on these relations. 'Strong' relations are preserved and 'weak' ones replaced. Not only does this approach deliver effective hit rates but also it decreases the communication

overhead in general as compared to other representative replacement algorithms such as LRU, LFU, and Greedy-dual size [4] when run against the DFStraces [8] from CMU.

We argue that our approach outperforms existing caching algorithms for two reasons. First, it minimizes the need for replacement and consequently the synchronization overhead. By a comprehensive simulation, we show that our approach produces the same or better hit rates than other algorithms, while at the same time the proposed algorithm results in only half of the number of replacements. Second, it takes the relationship between files into consideration while conventional approaches ignore them.

The rest of the paper is organized as follows. The design of the caching algorithm is presented in Section 2. Section 3 describes the trace files used in the performance evaluation. The details of implementation is reported in Section 4, followed by the performance evaluation in Section 5. Related work and concluding remarks are listed in Section 6 and Section 7 respectively.

2 Design

The basic idea of the proposed approach is motivated by the observation that file access patterns are not random, which are affected by the behavior of users and application programs. We believe that there is a semantic relationship between two files in a file access sequence for most of time. We classify this relationship into two categories, *inter-file relations* and *intra-file relations*. An inter-file relationship exits between two files A and B, if B is the next file opened following A being closed. A is called B's precursor. An intra-file relationship is said to exist between two files A and B if they are both open before they are closed. Our objective is to translate this relationship information into the notion of eviction index, based on which caching replacement can be performed. The relationship may be 'strong' or 'weak'. Our caching replacement algorithm preserves the 'strong' while replacing the 'weak'.

To define an inter-file relationship, we use the information obtained by studying DFStraces[8], which is the only public available distributed file systems traces, and our own file activity traces captured by system call interception, which was collected in 2004 [10]. We found that something common in both traces, a large portion of the files examined has a small number of unique precursors (precursors rather than successors are considered as this information is easy to obtain and manage). In some cases, 80% of the files have only one unique precursor as can be seen in Figure 1. Similar results have been observed in other study [6].

A heuristic parameter, $INTER_i$, is defined to represent the importance of a file i with respect to the *inter-file* relations with it's precursors. The bigger the importance the less likely it will be replaced. Therefore it is used as an eviction index by our caching algorithm. The importance of the file is determined by the following factors.

X_i - represents the number of times file i is accessed.

T_i - represents the time since the last access to file i.

T_j - represents the time since the last access to file j where j is a precursor of i.

Y_j - represents the number of times file j precedes file i.

$$INTER_i = \frac{X_i}{T_i + \sum_{j=1}^{n}(T_j - T_i)\frac{Y_j}{X_i}} \quad (1)$$

The importance of file i, as shown in Equation (1) is directly proportional to its access count and inversely to the time interval since its last access. The summation represents the strength of the inter-file relationship i has with its precursors.

The strength of the inter-file relationship between i and j depends not only on the recency of access of i or j represented by $T_j - T_i$ but also on the number of times j precedes i represented by Y_j. Therefore the bigger the weight $\frac{Y_j}{X_i}$, the more importance is given to the recency. Considering the case where file j (a popular precursor to file i meaning $\frac{Y_j}{X_i}$ is relatively large) has occurred more recently than file i, that is the recency $T_j - T_i$ would be negative, we can easily deduce that the summation value is reduced and the importance of file i is increased. This is exactly what we expect to happen, if j has been accessed recently, i is highly likely to be accessed next and should stay in the cache. Given this definition of the inter-file relationship, files with stronger relationships are given more importance than files with weaker relations.

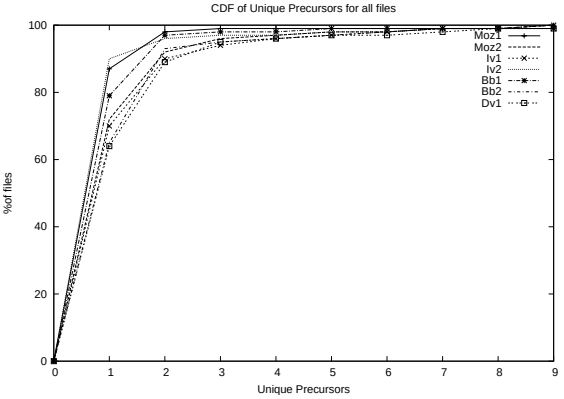

Figure 1: The cumulative distribution function of precursor counts of file access.

Next, we give the definition of the intra-file relationship, which exists between two files that open concurrently. An intra-file relationship is based on the concept of the *shared time*. Consider two files i and j that are both opened concurrently and i is closed before j is closed, then we define the *shared time of i with respect to j*, $S_{i,j}$, as the time between i's close and the later of the two opens as shown in Equation (2). The shared time of file i is calculated when file i is closed. The rationale behind this definition is files that have relatively large shared time are likely to share time in the future.

$$S_{i,j} = C(i) - MAX(O(i), O(j)) \tag{2}$$

where $O(i)$ and $C(i)$ are the open and close times of file i respectively and $C(i) < C(j)$.

A heuristic parameter, $INTRA_i$, is defined to represent the *un*importance of a file i based on it's *intra-file* relations, shown in Equation (3). This is opposite to the definition of the importance based on inter-file relations explained above, in the sense that the bigger the value the less important is the file. Here rather than depending on precursors, we define $INTRA_i$ based on the files that shared time.

$$INTRA_i = T_i + \sum_{j=1}^{n} (T_j - T_i) \frac{S_{i,j}}{S_{total}} \tag{3}$$

T_i - represents the time since the last access to file i.

T_j - represents the time since the last access to file j where j is open before i is closed.

$S_{i,j}$ - represents the shared time of file i with respect to file j where i is closed before j.

S_{total} - represents the total shared time with all files that are open before i is closed ($\sum_{j=1}^{n} S_{i,j}$).

The strength of the intra-file relationship between i and j depends on the recency of access of i and j represented by $T_j - T_i$, which is weighted by the relative shared time $\frac{S_{i,j}}{S_{total}}$. If file j has been accessed more recently than file i and $S_{i,j}$ is relatively large, $INTRA_i$ is reduced so that file i is less *un*important. On the other hand, if file j has been accessed much more before file i and $S_{i,j}$ is relatively large, the unimportance of i increases. This way, the intra-file relations are preserved in the cache using $INTRA_i$ as an eviction index.

Finally, it is straightforward to combine the inter-file and intra-file relationships to propose a combined caching replacement approach, represented as $BOTH_i$, as follows:

$$BOTH_i = \frac{X_i}{T_i + \sum_{j=1}^{n} (T_j - T_i) \frac{Y_j}{X_i} + \sum_{j=1}^{n} (T_j - T_i) \frac{S_{i,j}}{S_{total}}} \tag{4}$$

Next, before proceed to evaluate these approaches in one general context, we present the workload used in our evaluation.

Trace	Opens	Closes	Duration(hrs)	Files
Mozart1	25890	33953	49.43	709
Mozart2	93575	126756	162.83	1644
Dvorak1	122814	196039	121.75	4302
Ives1	41245	55862	75.70	247
Ives2	26911	36614	48.81	686
Barber1	30876	42155	52.12	725
Barber2	14734	20005	23.99	592

Table 1: Statistics of seven traces from DFSTraces.

3 Workload

Analysis of user behavior and file access patterns from *file system traces* help in making design decisions [12]. They also serve as workloads over which to test experimental systems. We used the DFSTraces [8] collected from Carnegie Mellon University as the workload to test our cache simulator. During the period from February '91 to March '93 the Coda project collected traces of all system call activity on 33 different machines. Analysis of seven traces files (two from Mozart, Ives and Barber each and one from Dvorak) were performed. The machine Barber was a server with the highest rate of system calls per second. Ives had the largest number of users, and Mozart was selected as a typical desktop workstation. Some of the statistics of the seven traces used are presented in the Table 1.

As these traces are a decade old it very likely that access patterns have since changed. We therefore developed our own system to collect traces of file access patters of users in a lab setting. We collected traces from four machines running Linux over periods ranging from a day to a week. This project is still a work in progress but we have already observed significant differences in access patterns and user behavior comparing with the DFS traces mainly because of the emergence of the Internet. What interestingly hasn't changed is the precursor counts of files. We still see a majority of files have at most one or two very 'popular' precursors among their individual set of precursors. This is again because file accesses aren't random.

4 Implementation

A simple client file system cache simulator was written to operate on the DFStraces as described in Section 3. We only consider the `open` and `close` system calls recorded in the traces. The cache itself is simulated using a hash table. We also maintain a list of closed and currently open files to keep track of changes to access count, last access time, precursors, and file size. The simulator goes through the trace looking for the `open` and `close` system calls. On encountering an `open`, we perform the following operations: First we check if space is available in the cache. If it is, an entry corresponding to the file that has

been opened is added to the *Open list* and *Close list*, and the file itself is added to the cache. The entry added into each list is different but share common information such as the filename, its size and the time it was opened. Every entry for closed file also maintains a precursor list and counts for each precursor in the list. Every new opened file is added to the end of the Close list. If it already exists in the Close List it's entries open time is updated with the new open time. The basic structure is shown in Figure 2. If the cache runs out of cache space, files need to be replaced to accommodate the new file. To prevent large files from pushing many small files out of the cache we define a $MaxFileSize$ threshold. Any file bigger than this threshold value isn't brought into the cache. We set this threshold as 30% of the total cache size. After this check is performed the main replacement policy is implemented. An eviction index is calculated for each file in the cache. Depending on the policy the file with the biggest or smallest index is removed from the cache. If this doesn't create enough space to accommodate the new incoming file the file with the next biggest/smallest index will be removed. This process repeats until enough space has been freed to accommodate the new file. Calculation of the index depends on the cache replacement policy.

To compare our approach with others in the same context, we have implemented seven different caching replacement algorithms as follows.

1. Round Robin (RR) - The eviction index is set to a constant value for all files. It provides a lower bound on performance. There is no reason to use any policy which performs worse than this one.

2. Least Recently Used (LRU) - The most frequently used algorithm in conventional caches. The eviction index represents the time since the last access to a file. File with the highest index is replaced.

3. Least Frequently Used (LFU) - This is based on the access counts of each file. The most popular files stay in the cache and the least popular files are replaced.

4. Greedy Dual-size (GDS) - This index is calculated based on the file size. Larger the file the smaller the index. File with the smallest index is replaced. We use the inflation value defined in [4] to keep frequently accessed large files in the cache.

5. INTRA - The eviction index is calculated based on intra-file relations as defined in Equation (3).

6. INTER - The eviction index is calculated based on inter-file relations as defined in Equation (1).

7. BOTH - The eviction index is calculated taking both inter-file and intra-file relations into consideration as defined in Equation (4).

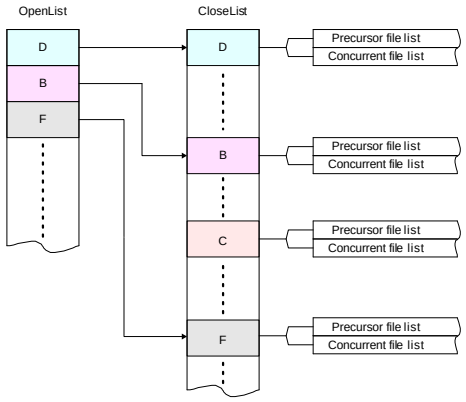

Figure 2: Basic data structure used by the replacement algorithm.

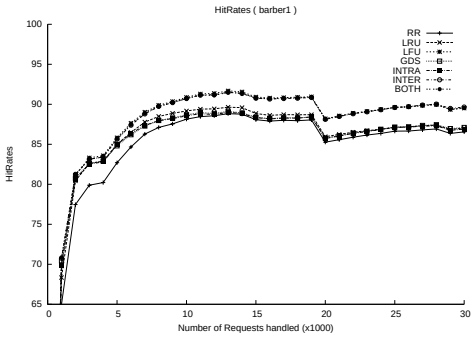

Figure 3: Hit rate variation over time for all algorithms for trace file barber1 (cache size=25KB)

Once the index for all the files are calculated, replacement is performed based on the eviction index until enough space is available. The new file is added to the cache. The performance of the cache is studied by varying its size.

5 Performance Evaluation

We evaluate our semantic-based approaches ($INTRA$, $INTER$, $BOTH$) by comparing against four conventional caching algorithms (RR, LRU, LFU and GDS), in terms of three parameters, *the cache hit rate*, *the byte hit rate*, and *the replace attempt*. The third parameter is used as an indicator of communication overhead in a low-bandwidth distributed environment, because each replacement at the client side necessitates a synchronization with a remote file server in a distributed file system. Due to space constraints only a small part of the evaluation is present here, more details can be found in technical report [10].

The first part of our evaluation compares the performance of all the algorithms using a particular trace

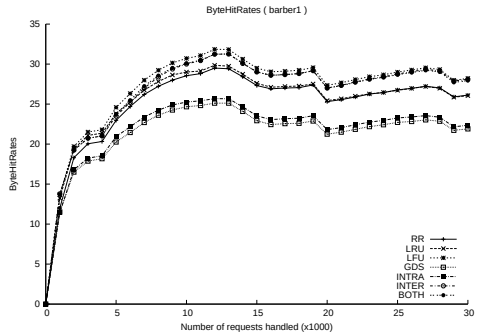

Figure 4: Byte hit rate variation over time for all algorithms for trace file barber1 (cache size=25KB)

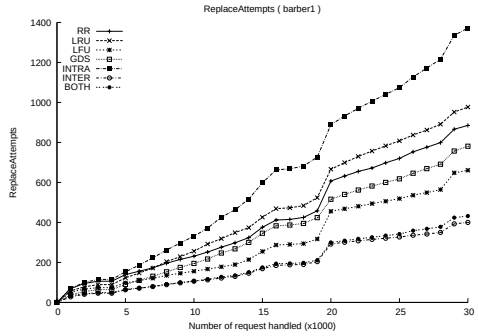

Figure 5: Replace attempts over time for all algorithms for trace file barber1 (cache size=25KB)

(`barber1`) and fixed cache size (25KB), as shown in Figure 3, Figure 4 and Figure 5. The x-axis of these figures is the number of file accesses has been processed, and the y-axes report the hit rate, the byte hit rate, and the replacement attempt, respectively. Each parameter is measured at the time when one thousand requests are handled. It can be seen that $INTER$ and $BOTH$ exhibit the highest hit rate and byte hit rates, and also the lowest number of replace attempts.

In general, in terms of hit rates, $INTRA$ performs very badly because of the very low number of intra-file relations existed in the DFS traces, which means file access is generally sequential in nature at that time. The DFS traces are a decade old and patterns could have changed which we hope the new traces we are working on will reflect. $INTER$ or $BOTH$ have the best hit rates in almost all cases [10], which validates our belief that file relations should be taken into consideraten while making caching replacement decisions.

The byte hit rates are much lower owing to a large number of small files, but we still see $INTER$ performing best. What is most interesting is the third parameter we measured, which gives us a measure of the utilization of the algorithm. Lower replace attempts indicate lower utilization. $INTER$ and $BOTH$ show the lowest number of replace attempts in some cases as much as half of that

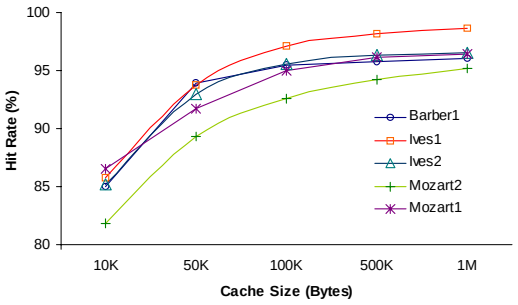

Figure 6: Hit rate of $BOTH$ for different cache sizes

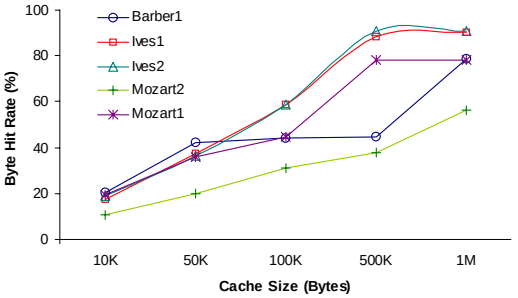

Figure 7: Byte hit rate of $BOTH$ for different cache sizes

of LRU and GDS, and at the same time produces better hit rates.

We feel a good replacement algorithm is one that is used as infrequently as possible to generate the same hit rates as conventional algorithms. Minimizing replacement requires the cache to have enough space every time a new file is encountered.

The next part looks at the performance of $BOTH$ in terms of the three parameters for different cache sizes. This is seen in Figure 6, Figure 7, and Figure 8, where the x-axis represents the changing cache size and the y-axis represents the corresponding change in the three performance metrics. As expected, the hit rate and byte hit rate increase with the increase of the cache size. Replace attempts drop with the increase of cache size. As expected, the hit rate and byte hit rate increase with the increase of the cache size. We also observe that beyond a point, the increase of cache size doesn't produce an equivalent increase in hit rate, as can be seen in Figure 6. Based on the traces used in our experiments, 100KB is a reasonable size for client-side cache using the proposed cache replacement algorithm. This is especially useful for mobile client with limited resources. Replace attempts drop with the increase of cache size. Theoretically if the cache is large enough communication with the server can be greatly reduced but practically large caches aren't feasible yet on resource constrained devices such as PDAs and cell phones.

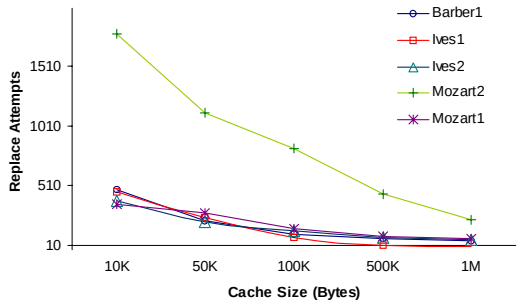

Figure 8: Replace attempts of *BOTH* for different cache sizes

6 Related Work

A large amount of work has been done in the area of cache replacement algorithms, as survey in [9]. Due to the space limit, only the efforts that take file relations into consideration are presented. Kuenning *et al.* [7] have done extensive work in the area of studying file system behavior in order to perform effective pre-fetching, where they define file relationships based on the overlap of file open and close events and hoard such related files. Other work performed by Amer *et al.* [1, 2] predicts future file access based on most recently occurring successors of files. The predictors developed are used to determine files to be pre-fetched. The crucial distinction between these approaches and our own is that rather than pre-fetching related files, we preserve these relationships in the cache. Pre-fetching not only involves a communication overhead but also requires a timely action.

7 Conclusions

We have presented a semantic-based cache replacement algorithm and shown that it performs better than conventional caching approaches in terms of the hit rate, the byte hit rate, and the number of replacements. Compared to prevalent replacement strategies that ignore file relations and communication overhead, this approach is more suitable for distributed file systems that operate across heterogeneous environments using resource constraint devices, especially with low-bandwidth connections.

References

[1] A. Amer and D. D. E. Long. Aggregating caches: A mechanism for implicit file prefetching. *Proceedings of the ninth International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2001)*, pp. 293-301, Aug. 2001.

[2] A. Amer, D. D. E. Long, J. F. Paris, and R. C. Burns. File access prediction with adjustable accuracy. *Proceedings of the International Performance Conference on Computers and Communication (IPCCC 2002)*, Apr. 2002.

[3] B. Callaghan and P. Staubach. NFS Version 3 Protocol Specification, rfc 1813, June 2000.

[4] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. *Proc. of the 1st USENIX Symposium on Internet Technologies and Systems (USITS'97)*, Dec. 1997.

[5] J. H. Howard, M. Kazar, S. Menees, d. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems* 6(1):51–81, Feb. 1988.

[6] T. M. Kroeger and D. D. E. Long. The case for efficient file access pattern modeling. *Proc. of the Hot Topics in Operating Systems (HotOS) VII*, Mar. 1999.

[7] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, Oct. 1997.

[8] L. B. Mummert and M. Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. *Software-Practice and Experience* 26(6):705–736, 1996.

[9] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.* 35(4):374–398, 2003.

[10] S. Santhosh. Factoring file access patterns and user behavior into caching design for distributed file system. Tech. Rep. MIST-TR-2004-013, Master of Science Thesis, Department of Computer Science, Wayne State University, Aug. 2004.

[11] W. Shi, H. Lufei, and S. Santhosh. Cegor: An adaptive, distributed file system for heterogeneous network environments. *Proceedings of the tenth International Conferences on Parallel and Distributed Systems*, July 2004.

[12] M. Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems* 14(2):200–222, 1996, `citeseer.nj.nec.com/spasojevic96empirical.html`.